Riku Leinonen

DATAPOHJAISUUS VIDEOPELEISSÄ

Opinnäytetyö
Kajaanin ammattikorkeakoulu
Luonnontieteiden ala
Tietojenkäsittelyn koulutusohjelma
Kevät 2013

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ
TIIVISTELMÄ

Koulutusala Luonnontieteiden ala	Koulutusohjelma Tietojenkäsittelyn koulutusohjelma
Tekijä(t) Riku Leinonen	
Työn nimi Datapohjaisuus videopeleissä	
Vaihtoehtoiset ammattiopinnot Peliohjelmointi	Ohjaaja(t) Joona Tolonen
	Toimeksiantaja
Aika	Sivumäärä ja liitteet 25

Tämä opinnäytetyö keskittyy ohjelmiston lähdekoodin ulkopuolisen datan reaaliaikaiseen käyttöön ja ajonaikaiseen lataamiseen ja lukemiseen videopeliohjelmistoissa. Työssä kerrotaan mitä datapohjainen pelinkehitys tarkoittaa ja miten peleissä käytetään ohjelmistokoodin ulkopuolisia tiedostoja. Tällaisen pelinkehityskäytännön käyttämisen syitä, sen tuomia hyötyjä sekä sillä tavoiteltavia pelin kehitysprosessin etenemistapoja kuvaillaan.

Toiminnallisessa osuudessa tässä opinnäytetyössä toteutettiin peliohjelma, joka lukee ja käyttää dataa ohjelman lähdekoodin ulkopuolisista tiedostoista. Datan lukemisen ja käyttämisen ohjelmoinnin toteutustapaa kuvaillaan ja perustellaan projektin ohjelmoijan näkökulmasta.

Kieli	Suomi
Asiasanat	Kajak3D, pelinkehitys, digitaaliset pelit
Säilytyspaikka	☒ Verkkokirjasto Theseus ☒ Kajaanin ammattikorkeakoulun kirjasto

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

THESIS ABSTRACT

School Business	Degree Programme Business Information Technology
Author(s) Riku Leinonen	
Title Data-Driven Game Development	
Optional Professional Studies Game Programming	Instructor(s) Joona Tolonen
	Commissioned by
Date	Total Number of Pages and Appendices 25

This thesis focuses on loading, parsing and usage of data files separate from source code in video game software development. It explains what data-driven game development is and how data files are used in games. It descripes the reasons for and the results of this kind of development paradigm.

The empirical part of this thesis is about designing and developing a video game that loads and uses data from files outside source code files of the game. The thesis descripes the design and programming process of the video game.

Language of Thesis	Finnish
Keywords	Kajak3D, game development, digital games
Deposited at	☒ Electronic library Theseus ☒ Library of Kajaani University of Applied Sciences

SISÄLLYS

1 JOHDANTO 1

2 DATAPOHJAINEN PELINKEHITYS 2

3 PELIOHJELMIEN DATATIEDOSTOT 4
3.1 Peliobjektien ominaisuudet 4
3.2 Karttatiedostot 4
3.3 Skriptit 5
3.4 Eri tiedostoformaatit 6
3.5 Esimerkki datatiedostojen käytöstä 7

4 PELIMOOTTORIT 9
4.1 Kajak3D 9

5 TOTEUTUS 11
5.1 Ohjelmakoodin rakenne 12
5.1.1 Tiedostojen avaus ja luku 13
5.2 Suunnittelu 14
5.2.1 Pelimekaniikkojen ulkoisille tiedostoille asettamat vaatimukset 15
5.2.2 Mitä dataa oli mahdollista ja kannattavaa ulkoistaa 15
5.2.3 Mitä tiedostomuotoa käytettiin ja miksi 17
5.3 Prototyyppien kehitys 18
5.4 Ohjelmistokoodin toimintalogiikka tiedostojen luvussa 18
5.4.1 Luetun datan tulkitseminen ja tarpeellisen tiedon poiminta 19
5.4.2 Datan lukemisen ja käytön jäsentely 20
5.5 Projektissa toteutetun pelin kohdealustan vaatimukset 21

6 POHDINTA 22

LÄHTEET 24

LIITTEET

SYMBOLILUETTELO

Bin	Binary, helposti tietokoneen luettavissa oleva, ihmiselle vaikeasti tulkittava tiedostomuoto.
MPQ	Mo'PaQ, Mike O'Brien Pack, tiedostojen pakkausformaatti.
OpenGL	Open Graphics Library, rajapinta grafiikan piirtämiseen.
OpenGL ES	Sulautetuissa ympäristöissä toimimaan suunniteltu OpenGL:n alijärjestelmä.
Singleton	Ohjelmoinnissa luokka, josta voidaan luoda vain yksi instanssi.
XML	Extensible Markup Language, merkintäkieli tiedon merkityksen kuvaamiseen tiedon sekaan.

1 JOHDANTO

Tämä opinnäytetyö keskittyy ohjelmiston lähdekoodin ulkopuolisen datan reaaliaikaiseen käyttöön ja ajonaikaiseen lataamiseen ja lukemiseen videopeliohjelmistoissa. Työssä kerrotaan mitä datapohjainen pelinkehitys tarkoittaa ja miten peleissä käytetään ohjelmistokoodin ulkopuolisia tiedostoja. Tällaisen pelinkehityskäytännön käyttämisen syitä, sen tuomia hyötyjä sekä sillä tavoiteltavia pelin kehitysprosessin etenemistapoja kuvaillaan.

Toiminnallisessa osuudessa tässä opinnäytetyössä toteutettiin peliohjelma, joka lukee ja käyttää dataa ohjelman lähdekoodin ulkopuolisista tiedostoista. Datan lukemisen ja käyttämisen ohjelmoinnin toteutustapaa kuvaillaan ja perustellaan projektin ohjelmoijan näkökulmasta.

Tällaisen toiminnallisuuden toteuttamisesta pelien kehityksessä oli hiukan aikaisempaa kokemusta, ja projektissa pystyttiinkin ottamaan käyttöön edellisten peliprojektien lähdekoodia. Aiheen valinta ja kiinnostus siihen pohjautui juuri näistä aikaisemmista kokemuksista datapohjaisen pelinkehityksen parissa. Se nähdään isona osana peliohjelmistojen kehitysprosessia, ja ansaitsee tarkempaa perehtymistä ja tutkimista.

2 DATAPOHJAINEN PELINKEHITYS

Datapohjaisessa peliohjelmistokehityksessä ihanteellisena tavoitteena on että ohjelmoijaa ei tarvita uuden sisällön kehittämiseen ja toteuttamiseen lainkaan. Kaikki sisältö pelissä on ulkoistettu lähdekoodista pois ulkoisiin tiedostoihin, joiden muokkaamiseksi ei tarvitse osata monimutkaista ohjelmointikieltä. Tämän ansiosta esimerkiksi suunnittelijat pystyvät itsenäisesti kehittämään uutta sisältöä peliin ja muokkaamaan vanhan sisällön ominaisuuksia ilman ohjelmoijan apua. Tämä vapauttaa ohjelmoijat tekemään omaa työtään, kun heidän ei tarvitse käyttää aikaa pelin sisällön lisäämiseen, ja säästää näin ohjelmiston kehityskuluissa ja –ajassa. (Bilas 2002.)

Datan ulkoistaminen ohjelmiston varsinaisesta lähdekoodista mahdollistaa pelisovellusten kuvaamien objektien ja ympäristöjen ja niiden ominaisuuksien muokkaamisen helposti ja käytännöllisesti. Tämä hyödyttää sovelluksen kehittäjää paitsi sovelluksen kehitysvaiheessa, myös sovelluksen valmistuttua sen muokkaamisen ja päivittämisen jälkikäteen ollessa helppoa. (Varanese 2002.)

Ohjelmakoodi ei ole peliä kehittäessä paras paikka aineistolle, jota joudutaan muuttamaan ja muokkaamaan useasti. Tällaisen aineiston erottaminen koodista erillisiin datatiedostoihin mahdollistaa tehokkaamman prosessin pelin sisällön ja toimintalogiikan muokkaamiseen. (Shumaker 2002.)

Pelin kehitys hidastuu ja kehittäjät turhatuvat helpommin, kun ohjelmoijat eivät ehdi ohjelmoida yhtä paljon kokonaan uusia ominaisuuksia vanhojen hienosäätämisen sijaan. Ohjelmistoon myös jää enemmän ohjelmointivirheitä ja ongelmia, kun vanhjojen ominaisuuksien korjaamiseen käytettävissä oleva aika vähenee. Näin pelin ominaisuuksien ja sisällön toteuttaminen pelkästään ohjelmistokoodilla voi vaikuttaa suoraan negatiivisesti pelituotteen loppukuluttajan käyttäjäkokemukseen. Suunnittelijat ja taiteilijat helposti väsyvät hitaaseen ominaisuuksien viilaamiseen ja jättävät täydellisten arvojen hakemisen kesken ja tyytyvät huonompaan lopputulokseen kuin mihin alun perin tähdättiin. Kaikkea tätä ehkäisemällä datapohjainen pelinkehitys parantaa projektin lopputuloksena syntyvän pelin laatua. (Brownlow 2004.) (Wilson 2002.)

Yksi hyöty sisällön ulkoistamisesta lähdekoodista muihin tiedostoihin on koodin uudelleenkäytön helpottuminen. Useasti iso osa yhdelle peliohjelmistolle uniikista sisällöstä on mah-

dollista toteuttaa ohjelmakoodin ulkopuolisella datalla. Kun lähdekoodissa on mahdollisimman vähän pelikohtaista sisältöä ja se sisältää vain pelinkehityksessä tarvittavan yleisen toiminnallisuuden, uutta projektia aloitettaessa vanhan pelin koodin on helpompi ottaa käyttöön ja muokata uuteen projektiin sopivaksi. Kun pelikohtaista sisältöä on koodissa vähän, myös jokaista peliä varten erikseen muokattavaa koodia on vähemmän ja suurempi osa koodista on uudelleenkäytettävissä sellaisenaan ilman aikaa vieviä muutoksia. (Wilson 2002.)

3 PELIOHJELMIEN DATATIEDOSTOT

Peliohjelmistojen käyttämä data voi olla monessa eri muodossa, esimerkiksi datan varastoimiseen käytetty tiedostomuoto voi vaihdella. Myös datan käyttötavat ja – tarkoitukset voivat vaihdella, kaikkia peliohjelmiston käyttämiä ulkoisia tiedostoja ei käytetä samalla tavalla samaan tarkoitukseen. Usein on tarkoituksenmukaista yhdenmukaistaa saman projektin käyttämä data sen käyttötarkoituksen mukaan siten, että samaan tarkoitukseen tuleva data tulee aina samassa muodossa, esimerkiksi yhden pelin kaikki tekstuuritiedostot samassa tiedostomuodossa. (Brownlow 2004.)

3.1 Peliobjektien ominaisuudet

Tieto peliobjektien alkuperäisistä ominaisuuksista säilötään usein pelinkehitystiimin tai sen yksittäisen ohjelmoijan kutakin projektia varten itse kehittämään muotoon. Peliobjektien ominaisuuksien säilömiseen ei siis ole alalla laajasti käytettyä standardimuotoa. Tämä tieto usein vielä muutetaan pelin julkaisua varten vaikeammin luettavaan ja tulkittavaan binaarimuotoon, vaikka se pelin kehityksen ajan pidettäisiin helpommin muokattavissa olevassa muodossa. (McShaffry 2009.)

3.2 Karttatiedostot

Usein karttadata tallennetaan tilan ja käytetyn muistin säästämiseksi kompaktissa muodossa, joka sisältää pelkästään viittauksia kartan sisältämiin objekteihin ja niiden käyttämiin resursseihin. Jos vaikkapa huoneeseen sijoittuva pelin kenttä käyttää esimerkiksi toistuvasti samaa huoneen lattialaatan pintakuviota, jokaista laattaa ei tallenneta erikseen, vaan kartta sisältää toistuvasti vain viittauksen yhteen tekstuuritiedostoon. Tämä sama tiedosto sitten ladataan muistiin yhdesti ja käytetään useamman kerran. Näin säästetään tilaa pelikoneen muistista, kun samaa dataa ei ladata sinne turhaan toistumaan useammin kuin kerran. (Adams 2002.)

Peleissä, joissa pelattavat tasot ovat laajoja, koko karttadataa ei voida kerralla ladata pelilaitteen muistiin, sillä se veisi muistista liikaa tilaa. Tällaisessa tapauksessa tiedosta voidaan ladata

laitteen muistiin pienempi osa kerrallaan, ja reaaliajassa vapauttaa muistista tarpeetonta dataa ja ladata tilalle uutta, tarpeellisempaa kenttätietoa. Useimmiten tällaisessa ratkaisussa tasosta ladataan muistiin pelaajan lähellä ja näkökentässä oleva osa, ja muistiin ladattua dataa vaihdetaan sen mukaan minne pelaaja liikkuu pelimaailmassa. (McShaffry 2009.)

3.3 Skriptit

Skriptit ovat pelimaailman ja sen sisältämien objektien käyttäytymistä ohjaavaa dataa. Skriptit kertovat peliobjekteille miten ne käyttäytyvät ja miten ne reagoivat toisiinsa ja pelaajan toimiin. Skriptit ovat ohjelmiston lähdekoodista erillään olevaa koodia, jonka ohjelmisto lataa ja usein myös kääntää ajon aikana. Skriptien koodia ei tarvitse kirjoittaa samalla ohjelmointikielellä jolla itse peli on toteutettu, ja usein ei kannatakaan. Skriptien kirjoittamiseen on olemassa tätä tarkoitusta varten kehitettyjä erityisiä skriptauskieliä. (Varanese 2002.)

Skriptikoodilla pystytään toteuttamaan toiminnallisuutta, joka ilman skriptejä tehtäisiin suoraan ohjelmiston omaa lähdekoodia muokkaamalla. Skriptit on usein kirjoitettu jollain ohjelmointikieltä muistuttavalla, joskin yksinkertaisemmalla skriptauskielellä. Tällaisen yksinkertaisen kielen avulla pelisuunnittelijat ja muut kenties vähemmän teknisesti orientoituneet pelinkehittäjät pystyvät toteuttamaan ja luomaan peliin suhteellisen monimutkaistakin sisältöä, ilman että varsinaisten ohjelmoijien tarvitsee käyttää aikaansa pelin sisällön tuottamiseen. Skriptaukseen voidaan käyttää valmiita, yleisessä käytössä olevia kieliä, mutta pelintekijät voivat myös kehittää omansa. Useasti toiminnallisuutta lähdekoodista ulkoistaessa kehittäjät jopa huomaamattaan keksivät oman skriptauskielensä, ulkoistetun toiminnallisuuden alkaessa yksinkertaisesta datasta ja monimutkaistuessa projektin edetessä ja kehittyessä skripteiksi. Skriptit ovat myös pelin omasta lähdekoodista erillisiä ja ne sekä ladataan peliin että käännetään ajon aikana, minkä seurauksena skriptiin tehty muutos ei vaadi lähdekoodin uudelleen kääntämistä. Tämän ominaisuuden ansiosta skriptien käyttö usein nopeuttaa pelinkehitysprosessia huomattavasti verrattuna siihen, että skripteillä toteutettu sisältö kirjoitettaisiin suoraan pelin omaan lähdekoodiin. (Adams 2002.) (Varanese 2002.)

Skriptit vaihtelevat monimutkaisuudessaan pelikohtaisesti. Joskus pelkillä skripteillä kyetään toteuttamaan hyvinkin monimutkaisia toimintoja ja ominaisuuksia pelissä muokkaamatta lähdekoodia lainkaan. Yleensä monimutkaisemmat järjestelmät ovat joustavampia ja mahdollistavat enemmän ominaisuuksia, mutta ovat toisaalta vaikeampia ja hitaampia oppia ja vievät

enemmän aikaa kehittää. Näistä syistä skriptausjärjestelmä tulisi valita kullekin projektille pelin tarpeiden mukaan. Usein pienemmät ja lyhyemmät projektit eivät vaadi monimutkaista skriptausta, ja sellaisen järjestelmän kehittäminen ja käyttäminen saattaa vain pidentää kehitysaikaa tarpeettomasti. Ennen toiminnallisuuden ulkoistamista lähdekoodista skriptiin olisi hyvä punnita kuinka paljon sisältö skripteillä toteutettaisiin, kuinka paljon resursseja skriptijärjestelmän kehittäminen vaatisi, ja näiden tietojen valossa miettiä toisiko skriptien käyttäminen ajallista säästöä ja olisiko se kannattavaa. (Adams 2002.)

3.4 Eri tiedostoformaatit

Pelinkehityksessä ei pelattavuuteen vaikuttavan datan suhteen ole käytössä laajalle levinneitä standardeja tiedostoformaatteja joita käytettäisiin yleisesti. Usein käytössä on kehitystiimin itse kehittämä formaatti, joka on räätälöity kutakin peliä varten peliohjelmiston tarpeiden mukaan. (McShaffry 2009.)

Yksi poikkeus tähän käytäntöön on XML-tiedostoformaatti, jota usein käytetään peliohjelmistojen tarvitseman datan varastoimiseen. Formaatti on helposti ihmisen luettavissa ja ymmärrettävissä, ja siten helposti ja nopeasti muokattavissa. Koska se on yleisesti käytössä muuallakin kuin peliohjelmistojen kehityksessä, sen lukemiseen ja muokkaamiseen on myös kattavasti saatavilla työkaluja. Tämä saattaa säästää ajassa, joka jouduttaisiin muuten käyttämään kehitystiimin oman tiedostomuodonluomisen ja muokkaamisen mahdollistaman ohjelmiston kehittämiseen. Joskus data muunnetaan XML-muodosta vaikeasti tulkittavaan binaarimuotoon ennen pelin julkaisua, jos muokattavuuden ei haluta olevan ohjelmiston käyttäjien saatavilla vaan käytettävissä pelkästään kehitysvaiheessa. (McShaffry 2009.)

Toinen yleisesti käytetty tapa säilöä pelien dataa on tekstitiedostojen käyttäminen. Menetelmän etuna on tekstitiedostojen selkokielisyys ja yleiskäyttöisyys. Tekstitiedostot pystytään avaamaan helposti ja nopeasti, ja niihin voi kirjoittaa ihmisten luettavissa olevalla kielellä, jolloin ei tarvita tietokoneohjelmaa tulkitsemaan tiedoston sisältöä. Nämä tiedostot ovat näin erittäin helposti muokattavissa. (DeLoura 2000.)

Myös teknisesti yleiskäyttöisiä tiedostomuotoja, kuten teksti- ja XML-tiedostoja käytettäessä pelin kehittäjien on silti päätettävä millä tavalla käytettävä data järjestellään näiden tiedostojen sisällä. Valittu tiedon organisointimenetelmä vaikuttaa suoraan siihen, millä tavalla dataa

luetaan tiedostosta, ja vaikuttaa siten sekä tiedostojen luettavuuteen, niiden tulkitsemisen helppouteen sekä peliohjelmiston lukemien tiedostojen tulkitsemisen tekniseen toteutukseen. (Harbour 2004.)

3.5 Esimerkki datatiedostojen käytöstä

Blizzard North –yhtiön kehittämä, Blizzard Entertainmentin julkaisema Diablo 2 käyttää runsaasti lähdekoodin ulkopuolista dataa ajon aikana. Kaikki pelin käyttämä data on pakattu tiedostoihin Blizzardin itse kehittämään MPQ-tiedostomuotoon (Wikipedia 2013). Yksi MPQ-tiedosto sisältää muunmuassa lukuisia ääni- ja kuvatiedostoja pelin käytettäväksi, ja runsaasti bin-muotoisia binääritiedostoja, joista peli lukee esimerkiksi tarvitsemansa tekstit ja numeeriset arvot. (Blizzard North 2000.)

	pet type	idx	group	basemax	warp	range	partysend	unsummon	automap	name	drawhp	icontype	baseicon	mclass1
1	none	0												
2	single	1		1	1		1		1					
3	valkyrie	2		0	1		1	1	1	StrUI4	1	1	valkarieicon	
4	golem	3		0	1		1	1	1	StrUI0	1	3	earthgolumicon	290
5	skeleton	4		0	1		1	1	1	StrUI1	1	2	skeletonicon	
6	skeletonmage	5		0	1		1	1	1	StrUI2	1	2	skeletonmageicon	
7	revive	6		0	1		1	1	1	StrUI3	1	2	revivedicon	
8	hireable	7		1	1		1		1		1		rogueicon	338
9	dopplezon	8		0				1	1		1			
10	invis	9		0	1									
11	raven	10		0	1					strUI5		2	raven	
12	spiritwolf	11	1	0	1		1	1	1	strUI6	1	2	wolf	
13	fenris	12	1	0	1		1	1	1	UIFenirsui	1	2	wolf	
14	totem	13		0	1		1	1	1	strUI14	1	3	oaksage	423
15	vine	14		0	1		1	1	1	strUI13	1	3	plaguepoppy	426
16	grizzly	15	1	0	1		1	1	1	strUI7	1	1	bear	
17	shadowwarrior	16		0	1		1	1	1	UiShadowUI	1	1	shadowassassin	
18	assassintrap	17		0		1								
19	pettrap	18		0		1								
20	hydra	19		0		1								

Kuvio 1. Diablo 2 –pelin käyttämää dataa sisältävä tekstitiedosto (Blizzhackers 2010).

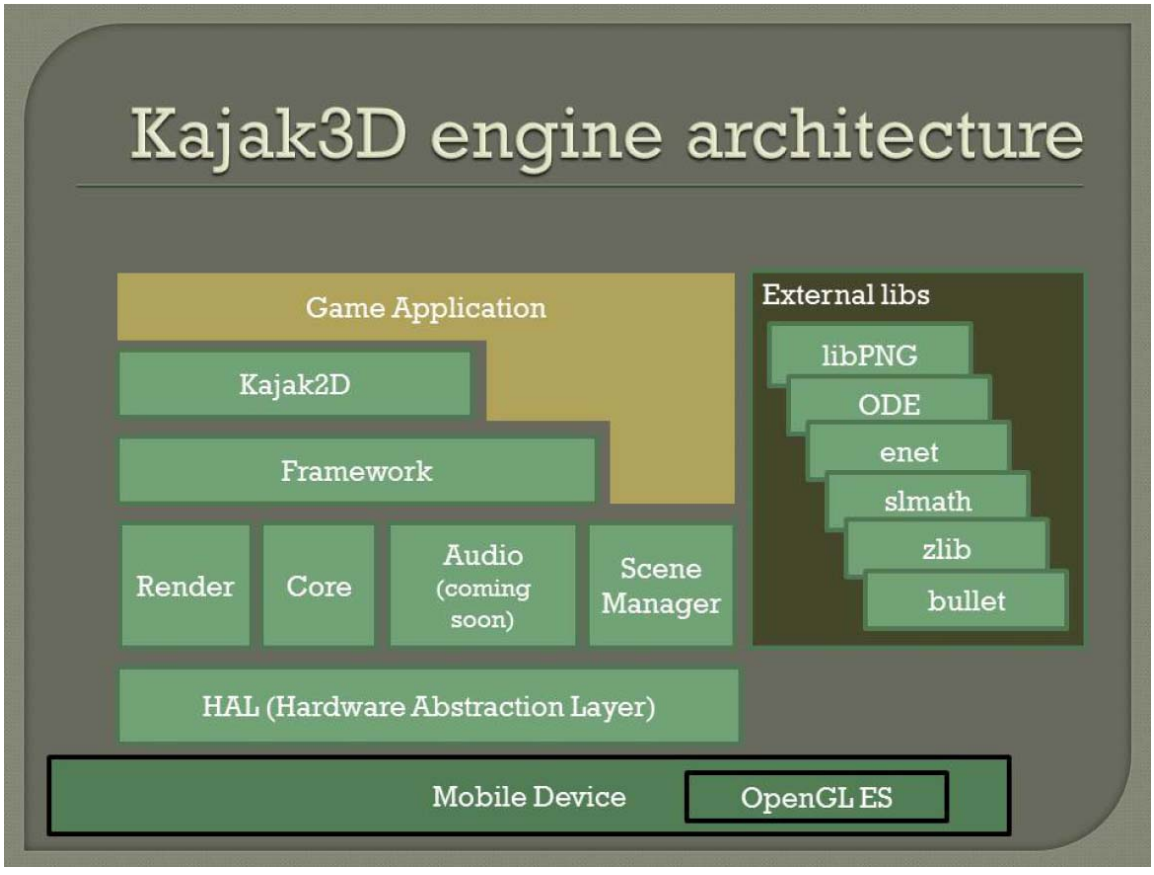
Bin-tiedostot eivät ole ihmissilmin luettavassa muodossa, joten nitä on vaikea muokata suoraan. Datapohjaisen kehityksen hyödyt eivät siis ole helposti ilmeiset pelin käyttäytymien perusteella. Diablo 2 kuitenkin osaa muuntaa oikealla tavalla jäsenneltyjä tekstitiedostoja (Kuvio 1.) bin-muotoon. MPQ-tiedostoissa on mukana jokaista .bin-tiedostoa kohden tekstitiedosto josta binäärimuoto on koostettu. Nämä tekstitiedostot voidaan purkaa pois MPQ-tiedostosta ja antaa pelille sitä käynnistettäessä komentoriviparametri, jota käyttäessä peli muuntaa nämä tekstitiedostot käynnistyessään bin-muotoon. Peli sitten käyttää näitä juuri rakennettuja binääritiedostoja MPQ-tiedostojen sisällä olevian bin-tiedostojen sijaan. Tällä

tavalla peli saadaan käyttämään helposti muokattavassa tekstitiedostomuodossa olevaa dataa. (Blizzard North 2000.)

4 PELIMOOTTORIT

Pelimoottori on peliohjelmiston kehittämisessä apuna käytettävä ohjelmiston osanen, jonka tarkoituksena on helpottaa ja nopeuttaa pelin tekemistä. Pelimoottori tarjoaa pelin kehittäjien käyttöön ohjelmallista toiminnallisuutta, jota kehittäjien ei näin tarvitse itse toteuttaa pelin omassa ohjelmistokoodissa. Usein pelimoottorin vastuulla on kuvan piirtäminen alustana käytetyn laitteen ruudulle, ja käyttäjän laitteelle tarjoaman syötteen lukeminen ja toimittaminen peliohjelmiston käyttöön. (Zerbst & Duevel 2004.)

4.1 Kajak3D



Kuvio 2. Kajak3D-pelimoottorin arkkutehtuurikaavio (Patana 2012).

Kajak3D on Kajaanin ammattikorkeukoulun kehittämä, pääasiassa mobiilialustoilla toimimaan suunniteltu pelimoottori, jonka arkkitehtuuri (Kuvio 2.) pohjautuu alun perin Java-

spesifikaatioon. Mobiiliympäristöjen lisäksi se toimii myös PC- ja pelikonsolialustoilla. Kajak3D:a käyttäviä ohjelmistoja on tehty pääasiassa iOS-, Android- ja Windows-alustoille. Moottorin pääasiallinen tarkoitus on helpottaa sovelluksien luomista usealle alustalle ilman suuritöisiä muutoksia ohjelmiston lähdekoodiin. Moottorin piirto-ominaisuudet käyttävät OpenGL- ja OpenGL ES -rajapintoja. Kajak3D on Kajaanin ammattikorkekoulussa opetuskäytössä, ja myös ulkopuolisten toimijoiden saatavilla avoimen lähdekoodin LPGL-lisenssin alla. (Patana 2012.)

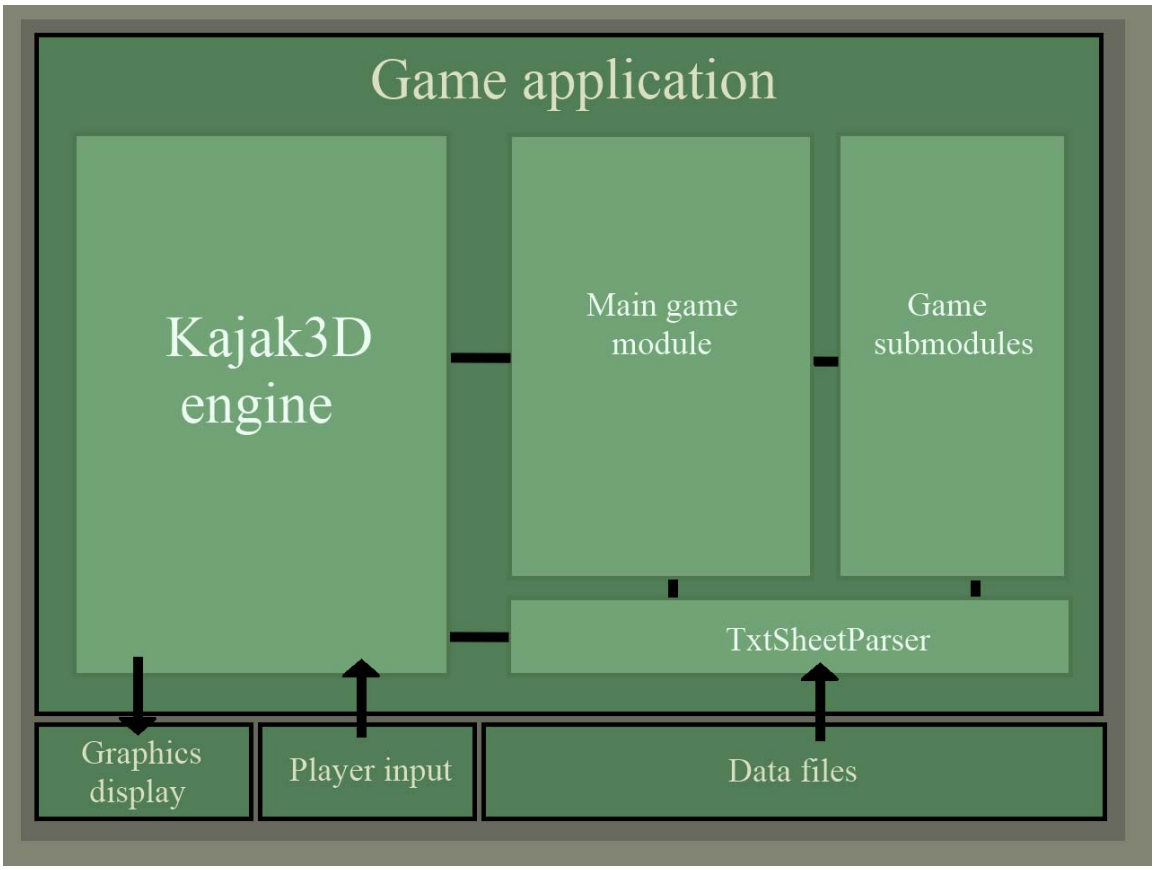
5 TOTEUTUS

Toiminnallisena osuutena tässä opinnäytetyössä oli toteuttaa peliohjelmisto Android-käyttöjärjestelmää käyttäville mobiilialustoille. Pelin tuli jollain tavalla ladata ja hyödyntää lähdekoodin ulkopuolista dataa ajon aikana. Projektin kesto oli 7 viikkoa.

Opinnäytetyötä varten toteutetussa peliprojektissa päädyttiin tekemään pelisovellus Android-käyttöjärjestelmälle Kajak3D-pelimoottoria hyödyntäen. Pelissä pelaaja ohjaa alusta ja taistelee vihollisaluksia vastaan. Vihollisalusten ominaisuudet määräytyvät satunnaisesti joka pelikerralla. Oleellinen osa pelin sisältöä ovat vihollisalusten ominaisuudet ja niiden vaihtelevuus. Kyseiset ominaisuudet oikein toteuttamalla pelistä saadaan mielenkiintoisempi ja vähemmän yksitoikkoinen. Nämä ominaisuudet päätettiin toteuttaa lähdekoodin ulkopuolisten tiedostojen avulla, jotta niiden lisääminen ja muokkaaminen olisi helppoa. Tällä tavoin lisäsisällön tuottaminen peliin helpottuisi ja nopeutuisi. Tavoitteena projektia aloittaessa oli ulkoistaa lähdekoodista pois mahdollisimman paljon pelin sisällöstä.

Projektin ohjelmoinnin osuus eli lähdekoodin kirjoittaminen jaettiin kolmeen osioon, suunnitteluun, prototyyppien luontiin ja testaukseen sekä lopullisen peliohjelmiston toteutukseen. Suunnittelun tarkoituksena oli ehkäistä aikaa vieviä ongelmatilanteita keskellä projektia. Prototyyppien avulla ideoita ja ratkaisuja pystytään nopeasti testaamaan ja havaitsemaan niiden mahdolliset puutteet ennen kuin investoidaan paljon aikaa lopullisen ohjelmakoodin kirjoittamiseen. Näiden kahden ennakkovaiheen avulla tähdättiin nopeampaan ja parempaan lopputulokseen.

5.1 Ohjelmakoodin rakenne



Kuvio 3. Toteutetun peliohjelman arkkitehtuurikaavio.

Projektissa käytettiin C++-ohjelmointikieltä ja Kajak3D-pelimoottoria. Nämä valittiin käytettäväksi koska kehittäjillä oli niiden käytöstä aikaisempaa kokemusta, ja valittu pelimoottori mahdollisti pelin kehittämisen useammalle eri alustalle nopeammin ja vaivattomammin kuin vaihtoehtoiset ratkaisut.

Pelin ohjelmoinnissa päätettiin käyttää kontrollerimoduuleihin jaettua komponenttipohjaista järjestelmää (Kuvio 3.), joka koostuu erillisistä itsenäisistä moduuleista. Moduulit hallinnoivat yhtä osa-aluetta kaikkien peliobjektien käyttäytymisestä, esimerkiksi objektien liikuttaminen on yhden moduulin hallinnoimaa, ja ammuksiin törmääminen toisen. Kullakin moduulilla on lista sen kontrolloimista objekteista ja kuhunkin objektiin liitetystä komponentista, joka määrittelee millä tavalla objekti käyttäytyy. Moduulit tietävät toisistaan vain tarvittaessa, mikä helpottaa moduulien muokkaamista, kun yhtä moduulia muutettaesa ei tarvitse samalla muokata montaa muuta siitä riippuvaista osaa ohjelmistosta.

Useat näistä moduuleista tarvitsevat ulkoisissa datatiedostoissa säilöttyä tietoa, ja niiden tulee jotenkin päästä käsiksi siihen. Jos kukin moduuli sisältäisi oman logiikkansa tiedostojen avaamiseen ja lukemiseen, se johtaisi huomattavaan määrään ohjelmakoodin kopioimista paikasta toiseen, ohjelmointivirheiden kertautumista, ja kyseisen logiikan muokkaamisen hitauteen ja yleiseen hankaluuteen.

Suunnittelussa keskityttiin järjestämään datan luku lähdekoodin ulkopuolisista tiedostoista kätevästi, nopeasti ja tehokkaasti. Tiedostojen luku haluttiin keskittää yhteen paikkaan, kuitenkin siten että kaikilla moduuleilla jotka tarvitsivat pääsyn dataan tulisi olla helppo väylä päästä käsiksi siihen. Samalla pääsyä tiedostoihin haluttiin silti rajoittaa, jotta joka moduulilla ei olisi tarpeettomasti mahdollisuutta avata ja lukea dataa. Tiedostojen avaus ja luku on suhteellisen hidasta, joten sitä ei missään tapauksessa haluttu suorittaa kesken pelin intensiivisen osuuden, jossa ohjelmiston realiaikainen nopea toimivuus olisi tärkeää.

Nämä kriteerit mielessä pitäen suunnitelmassa päädyttiin keskittämään kaikki tiedostojen avaus ja luku yhteen luokkaan. Tästä luokasta luodaan yksi instanssi ohjelmiston käynnistyessä, ja jokainen tiedostoista dataa tarvitseva moduuli saa parametrina osoittimen tähän instanssiin. Valitun lähestymistavan heikkoutena osoitinta tähän yksittäiseen instanssiin jouduttaisiin kuljettamaan parametrina joskus syvällekin metodikutsujen hierarkiassa, ja se jouduttaisiin säilömään eri moduuleita hallinnoivalla ylätasolla pelin päämoduulissa, vaikka sitä ei tarvitsisi varsinaisesti käyttää siellä. Ylemmältä tasolta osoitin instanssiin pystyttäisiin jakamaan alemman tason moduuleille. Tämä potentiaalisesti luo myöhemmin jatkokehityksessä hämmennystä sen suhteen, miksi ylemmän hierarkian hallintomoduulilla on pääsy tiedostojen lukuun vaikka se ei sitä itse käytä eikä tarvitse. Nämä haittapuolet todettiin hyväksyttäviksi ja pienemmiksi kuin harkinnassa olleiden vaihtoehtoisten ratkaisujen vastaavat.

5.1.1 Tiedostojen avaus ja luku

Kehitetyn pelin ohjelmistokoodissa ulkoisten tiedostojen lukemisessa keskeisessä osassa on TxtSheetParser-parseriluokka, jonka tehtävänä on avata tekstitiedostoja ja lukea niiden sisältö talteen pelin muistiin. Dataa tarvitsevat moduulit antavat tälle luokalle tiedostopolun luettavaan tiedostoon, sekä viittauksen taulukkoon johon luettava data voidaan säilöä. Parseri ei itse säilytä viittauksia näiden tiedostojen dataan, vaan täyttää sille parametrina annetun taulu-

kon lukemallaan datalla. Tiedoston dataa tarvitsevan moduulin tulee itse huolehtia viittauksien säilyttämisestä.

Tällaista luokkaa päädyttiin käyttämään muun muassa eristämään pelin toimintalogiikkaa hallinnoivat moduulit niiden tarvitsemasta datasta ja sen lukemisesta. Näiden toimintamoduulien ei tarvitse tietää muuta kuin sen tiedoston nimi, josta niiden haluama data luetaan. Tarvittaessa tarvittavat tiedostonimetkin pystytään lukemaan realiajassa tiedostosta, mutta silloin sen tiedoston nimen joka sisältää muiden datatiedostojen nimet on oltava moduulin tiedossa.

Tiedoston sisältämän datan läpikäyminen ja sen tulkitseminen siten, että juuri haluttu data löydetään kaiken tiedoston sisältämän datan joukosta on usein monimutkainen ja lukuisia koodirivejä vaativa prosessi. Tiedostosta joudutaan seulomaan oleellinen, pelin tarvitsema sisältö käyttämällä tiedoston sisältämiä viittauksia, jotka kertovat missä kohtaa tiedostoa mikäkin tieto sijaitsee ja jotka ovat sinänsä epäolennaisia pelin toiminnallisuuden kannalta. Tämä pitkällinen operaatio haluttiin piilottaa pois pelilogiikkaa hallinnoivista moduuleista metodikutsun taakse yhteen luokkaan, joka hallinnoi tiedostojen lukua ja parsimista. Samalla tämnä parsimisprosessi saatiin eristettyä yhteen paikkaan, jossa sitä on helppo tarvittaessa muokata

5.2 Suunnittelu

Pelin ohjelmakoodin suunnitteluvaiheessa käytiin läpi ohjelmiston koodille asettamat vaatimukset ja suunniteltiin niiden pohjalta koodille karkea luokkakaavio. Ohjelmiston vaatimien ominaisuuksien toteuttamiseen tarvittavat luokat ja niiden sisältämät metodit käytiin läpi. Tarkoituksena ei ollut tässä vaiheessa pysyvästi päättää koodin toteutuksesta mitään, vaan suunnitelmaa käytettiin karkeana ohjenuorana nopeuttamaan kehitystä sekä ennakoimaan mahdollisia ongelmatilanteita.

Suunnitteluvaiheessa myös pyrittiin kartoittamaan projektin yksilölliset tarpeet teknisten ratkaisujen osalta. Ulkoistettaessa dataa lähdekoodista tekniset ratkaisut kannattaa tehdä kunkin projektin tarpeiden mukaan. Käytettäviä tiedostoformaatteja ja datan käyttökohteita valitessa haluttiin ottaa huomioon esimerkiksi kehittäjien aikaisempi kokemus ja mieltymykset, pelin pelimekaniikat sekä pelin kohdealustan vaatimukset. Tämä tarkoitti, että peliin haluttu pelattavuus asetettiin etusijalle ja pyrittiin valitsemaan käytännön toteuksessa käytetyt menetelmät

siten, että ne tukisivat tätä pelattavuuden tavoitetta missä vain mahdollista. Kunkin pelin toiminnalliset ominaisuudet monesti asettavat vaatimuksia peliohelmiston tekniselle toteutukselle, ja yhdelle pelille hyvä ratkaisu ei välttämättä toimi hyvin toisessa. Esimerkiksi datan tallennukseen käytettävää tiedostomuotoa valittaessa on hyvä miettiä, minkälaisen datan tallennusta kehitettävä peli vaatii, ja mikä tiedostomuoto olisi hyvä tälle datalle.

5.2.1 Pelimekaniikkojen ulkoisille tiedostoille asettamat vaatimukset

Opinnäytetyötä varten toteutetussa projektissa kehitettävä peli sisälsi olennaisena osana pelikonseptia satunnaisesti luotuja vihollisia. Vihollisalukset liikkuivat ruudulla ja laukaisivat pelaajalle vahingollisia ammuksia, jotka pystyivät vaikuttamaan pelaajaan eri tavoilla osuessaan pelaajan alukseen. Nämä viholliset päätettiin toteuttaa siten, kullekin vihollisen satunnaiselle ominaisuudelle määriteltiin etukäteen joukko arvoja joiden väliltä valitseminen suoritettiin.. Valinnan piirissä olevien arvojen joukko määräytyi kyseiselle viholliselle säädetyn vaikeustason mukaan. Esimerkiksi vihollisen ammusten lentorata, ammusten määrä ja ammusten tyyppi valittiin satunnaisesti ennalta määrätyistä vaihtoehdoista, ja helpoimmilla vaikeustasoilla pelaajan pelissä onnistumista eniten vaikeuttavat arvot oli rajattu pois. Vihollisten luonti ja niiden ominaisuuksien valinta suoritettiin dynaamisesti ajon aikana.

Vihollisten satunnaisesti valittujen ominaisuuksien haluttiin olevan helposti muokattavissa sekä kehityksen aikana että pelin julkaisemisen jälkeen. Jos jokin vihollistyyppi osoittautuisi esimerkiksi liian vaikeaksi, sitä haluttiin pystyä helpottamaan nopeasti ja vaivattomasti. Samoin uuden sisällön lisäämisen peliin julkaisun jälkeen piti onnistua ilman suurta työmäärää. Vihollisten käyttöön piti pystyä julkaisun jälkeenkin esimerkiksi lisäämään kokonaan uusi ammustyyppi tai määrittelemään ammuksille uusi lentorata, ja uusien vihollisten lisäämisen peliin tuli olla mahdollisimman vaivatonta.

5.2.2 Mitä dataa oli mahdollista ja kannattavaa ulkoistaa

Edellä määritettyjen vaatimusten täyttämiseksi vihollisten satunnaiset ominaisuudet päätettiin toteuttaa siten, että ne luettiin lähdekoodin ulkopuolisista tekstitiedostoista. Tämä mahdollisti ominaisuuksien helpon muokkaamisen jälkikäteen. Uusien arvojen lisääminen joukkoihin,

joista peli valitsi satunnaisten ominaisuuksien arvot, onnistui myös helposti. Tällä tavalla mahdollistettiin sekä lisäsisällön kehittäminen peliin että ominaisuuksien muokkaus ja esimerkiksi mahdollisten ongelmien korjaus nopeasti ja vaivattomasti vielä julkaisun jälkeen.

Tämän lisäksi myös pelissä alusten käyttämät erilaiset asetyypit sekä näiden aseiden käyttämät erityyppiset ammukset toteutettiin tallentamalla niiden ominaisuudet tekstitiedostoihin. Tällä tavalla toteutettuna aseiden ja ammusten ominaisuuksien muokkaus onnistui nopeasti ja helposti. Jokaisella aseella oli tekstitiedostossa esimerkiksi viittaus aseen käyttämään ammukseen. Jos aseen käyttämää ammustyyppiä haluttiin vaihtaa, riitti, että tekstitiedostosta vaihdettiin tämä viittaus johonkin toiseen ammukseen, eikä pelin lähdekoodia tarvinnut kääntää uudestaan.

Täysin uusien ammusten ja aseiden lisääminen oli vaivalloisempaa, sillä kokonaan uudentyyppinen toimintalogiikka näille komponenteille vaati tukea lähdikoodista. Jo olemassaolevien aseiden ja ammusten ominaisuuksien yhdistely uudenlaisiksi kokonaisuuksiksi oli mahdollista, mutta kokonaan uusia ase- ja ammustyyppejä peliin ei pystynyt lisäämään tekemättä muutoksia lähdekoodiin. Esimerkiksi täysin uudenlaista lentorataa käyttävä ammus tarvitsi ensin halutun lentoradan toteuttamisen ohjelmakoodissa. Sen sijaan vanhan, jonkun muun ammuksen jo käyttämän, valmiiksi toteutetun lentoradan käyttäminen yhdessä jo olemassaolevan ammustyypin kanssa oli mahdollista pelkästään ulkoisia tekstitiedostoja muokkaamalla. Tämän vuoksi peliin haluttiin sisällyttää toteutus mahdollisimman monelle erilaiselle sisällölliselle ominaisuudelle, vaikka niitä kaikkia ei pelin ensimmäisessä versiossa käytettäisikään. Näin niitä voitaisiin tarpeen vaatiessa ottaa käyttöön tulevaisuudessa lähdekoodia muokkaamatta pelkästään datatiedostoja muuttamalla, ja muutosten jakaminen käyttäjille pelin julkaisemisen jälkeen olisi helpompaa ja vaivattomampaa.

Kuvatunlaisen täysin uuden sisällön lisääminen peliin ulkoisten tiedostojen avulla lähdekoodia muokkamatta olisi teknisesti ollut mahdollista ottamalla käyttöön jonkinlainen skriptauskieli, jolla peliobjekteille olisi pystynyt määrittelemään uudenlaisia toimintoja ja käyttäytymismalleja. Skriptien käyttöönotto kuitenkin nähtiin liian aikaa vieväksi ja vaativaksi operaatioksi projektin tarpeisiin nähden. Kehitäjillä ei ollut aikaisempaa kokemusta skriptien käytöstä, ja projektin pienen mittakaavan ja sisällön vähäisen määrän ansiosta kyseisenlaisen uuden sisällön/ lisäämisen lähdekoodia muokkaamalla ja uudelleen kääntämällä arvioitiin vievän vähemmän aikaa kuin skriptauksen opettelun ja lisäämisen peliin. Projektin lyhyen aika-

taulun ei arvioitu mahdollistavan skriptijärjestelmän rakentamista, eikä sellaisen järjestelmän tarjoamien hyötyjen olevan resurssikustannusten arvoisia.

Varsinaiseen pelattavuuteen vaikuttamattomasti myös tiedostopolut pelin käyttämiin resurssitiedostoihin luettiin ajon aikana tiedostosta, esimerkiksi käytettyjen ääni- ja kuvatiedostojen sijainnit ladattiin muistiin tällä tavalla. Tällaista ratkaisua ei koettu todella tärkeäksi eikä sitä priorisoitu kovin korkealle, mutta se oli kuitenkin helppo ja nopea toteuttaa, sillä järjestelmät tiedostojen avaamiseen ja lukuun jouduttiin toteuttamaan muutenkin. Tämän menetelmän käyttö mahdollisti kyseisten resurssien tiedostonimien ja –sijantien vaihdon ilman lähdekoodin muokkaamista, jos vaikkapa pelin taustamusiikki haluttiin vaihtaa jossain vaiheessa projektia.

5.2.3 Mitä tiedostomuotoa käytettiin ja miksi

Tämän opinnäytetyön toiminnallisessa osuudessa toteutettua peliprojektia varten päädyttiin käyttämään lähdekoodin ulkopuolisten datatiedostojen tiedostoformaattina tekstitiedostoja. Tähän ratkaisuun päädyttiin projektin sen vuoksi, että projektin ohjelmoija omasi eniten aikaisempaa kokemusta tällaisen formaatin käyttämisestä vastaavanlaiseen tarkoitukseen. Lisäksi käytetty pelimoottori, Kajak3D, mahdollisti tekstitiedostojen helpon lataamisen ja käsittelyn. Tiedostomuodon käyttöä varten myös pystyttiin ottamaan käyttöön lähdekoodia aikaisemmista projekteista, joissa oli myös käytetty tekstitiedostoja ohjelmiston tarvitseman datan tallentamiseen. Nämä tekijät nopeuttivat pelin kehittämistä, eikä syytä ottaa käyttöön jotain muuta tiedostomuotoa nähty. Tekstitiedostojen helppolukuisuutta ja muokattavuutta myös pelaajien osalta ei koettu haittatekijäksi, sillä kohdealustana toimineen Android-käyttöjärjestelmän vaatimusten vuoksi koko ohjelmisto ja sen sisältämät tiedostot piti joka tapauksessa pakata binaarimuotoiseen apk-tiedostomuotoon, jonka kautta pelaajat eivät helposti pääsisi käsiksi pelin sisältämään dataan.

Tiedon organisointi näiden tekstitiedostojen sisällä päätettiin hoitaa järjestämällä data taulukkomuotoon. Tällaista ratkaisua oli käytetty kehittäjien aikaisemmissa ohjelmistoissa, joten siitä oli aiempaa kokemusta, ja sen käyttöönottoa varten pystyttiin uudelleenkäyttämään koodia näistä aiemmista projekteista. Käytetty rakenne muistuttaa paljon Blizzard Northin Diablo 2 –pelin käyttämää tiedostorakennetta. Taulukko koostuu riviin ja sarakkeisiin järjestetyistä tietueista, ja jokaisella rivillä ja sarakkeella on oma otsikkonsa. Ohjelmisto löytää tar-

vitsemansa datan etsimällä tiedostosta sen rivin ja sarakkeen jolta tiedon on määritelty löytyvän. Nämä rivien ja sarakkeiden otsikot pystytään halutessa myös määrittämään jonkin tiedoston sisällä, tai kirjoittamaan suoraan ohjelmiston lähdekoodiin projektin tarpeiden mukaan.

5.3 Prototyyppien kehitys

Prototyyppien tarkoituksena on antaa nopeasti käsitys ideoiden ja ratkaisujen toimivuudesta käytännössä. Tätä opinnäytetyötä varten tehdyssä projektissa kehitettiin yksi prototyyppi, johon yksitellen lisättiin uusia ominaisuuksia testattavaksi siten, että prototyypin laajuus ja monimutkaisuus kasvoi projektin edetessä. Prototyyppiä kehitettiin siten että kulloinkin testattava ominaisuus saatiin toimimaan mahdollisimman nopeasti, välittämättä esimerkiksi lähdekoodin huollettavuudesta tai helppolukuisuudesta. Prototyypin avulla toimiviksi todetut ominaisuudet toteutettaisiin jälkeenpäin ohjelmiston lopulliseen tuotantoversioon, jota kehittäessä tuotannon nopeus ei olisi ensisijainen tekijä, ja ohjelmiston helppoon huollettavuuteen ja muihin pitkän tähtäimen tekijöihin panostettaisiin enemmän. Tämän menetelmän tarkoituksena on nopeasti todeta mitkä ideat toimivat käytännössä ja mitkä eivät, jolloin huonot ratkaisut saadaan karsittua nopeasti pois ja aikaa ei käytetä niiden edelleen kehittämiseen.

Tällainen prototyyppien käyttö ohjelmiston kehityksessä valittiin myös ohjelmoinnin nopeuttamiseksi ja helpottamiseksi. Kun ohjelmoija ensimmäistä kertaa pelin ominaisuutta toteuttaessaan tietää sen olevan osa lyhytikäistä prototyyppiä testausta varten, hänen ei tarvitse turhaan käyttää aikaa koodin uudelleenkäytettävyyteen ja muihin pitkän aikavälin ominaisuuksiin. Tämä nopeuttaa koodin kirjoittamista huomattavasti. Peliä varten ideoidun ominaisuuden toimivuus käytännössä saadaan testattua erittäin nopeasti, ja huonojen ideoiden hiomiseen ei käytetä turhaan aikaa. Tieto siitä, ettei nyt kirjoitettavaa koodia tarvitse käyttää ja sen kanssa työskennellä tulevaisuudessa, helpottaa ohjelmoijan työtaakkaa.

5.4 Ohjelmistokoodin toimintalogiikka tiedostojen luvussa

Datan luku tiedostoista haluttiin siis hoitaa kätevästi, nopeasti, ja tehokkaasti. Tärkeänä kriteerinä koodissa pidettiin muokattavuutta ja huollettavuutta. Tiedostojen avaus ja luku halut-

tiin hoitaa keskitetysti yhdessä paikassa, jotta muutoksia ja mahdollisten ongelmien korjaustoimenpiteitä ei tarvitsisi hajauttaa ympäri lähdekoodia. Tämä myös helpottaa koodin uudelleenkäytettävyyttä, jolloin tätä projektia varten kirjoitettua koodia voitaisiin mahdollisesti käyttää myös muissa ohjelmistoissa tulevaisuudessa, ja myös helpommin käyttää aiemmin kirjoitettua koodia tässä projektissa..

5.4.1 Luetun datan tulkitseminen ja tarpeellisen tiedon poiminta

TxtSheetParser-luokka hoitaa luetun datan tulkitsemisen ja sisältää metodit yksittäisten tietueiden etsimiseen taulukkomuodossa tallennetusta datasta. Moduulit, joilla on taulukkoon tallennettuna tiedostoista luettua dataa, antavat parserille parametrina viittauksen taulukkoon, sekä tekstimuodossa taulukon sarakkeen ja rivin otsikon. Parseri etsii viitatusta taulukosta rivin jolla on annettu otsikko, ja palauttaa moduulille kyseiseltä riviltä sen tietueen sisällön, jonka sarakkeen otsikko täsmää annettuun.

Pelin datassa taulukoiden rivien otsikkona käytetään usein kunkin tyyppiselle peliobjektille annettua tunnistetta, joka määrittelee minkälaiseen objektiin rivin data liittyy. Käsitellessään peliobjektia ohjelmiston moduuli sitten käyttää kyseisen objektin tunnistetta sen rivin nimenä, jolta tietoa haetaan. Joissain tapauksissa moduulit lukevat tarvittavan sarakkeen nimen jonkin toisen dataan liittyvän tiedoston riviltä, jonka otsikkona käsiteltävän objektin tunniste on. Kunkin moduulin lähdekoodiin on suoraan määritelty minkä nimiseltä sarakkeelta modulin tarvitsema tieto löytyy.

TxtSheetParser-luokka sisältää myös metodeja tiedostoista ladatun datan muunlaiseen käsittelyyn, kuten yksittäisen sarakkeen eristämiseen kokonaisesta datataulukosta. Luokka osaa myös tallentaa sille taulukkomuodossa annetun datan takaisin tekstitiedostoon. Jälkikäteen ajateltuna tämän taulukkomuotoisen tiedon tulkitsemisen ja selaamisen, sekä tiedostojen avauksen ja käsittelyn olisi todennäköisesti kannattanut erottaa toisistaan sen sijaan, että molemmat nyt käytetyn järjestelmän mukaisesti toteutetaan samassa luokassa.

5.4.2 Datan lukemisen ja käytön jäsentely

Kaikki ohjelmiston moduulit, joiden tarvitsee päästä käsiksi tekstitiedostoihin tallennettuun dataan, sisältävät osoittimen TxtSheetParser-luokan instanssiin. Tiedostojen avaaminen ja lukeminen muistiin kiintolevyltä on suhteellisen hidasta, joten kyseinen operaatio suoritetaan joko aivan alussa heti peliohjelmiston käynnistyessä tai pelin päävalikosta pelin toimintavaiheeseen siirryttäessä lataustauon aikana. Itse realiaikaisen pelin toimintavaiheen ollessa käynnissä tätä operaatiota ei suoriteta, silloin dataa tarvitsevat moduulit vain käyttävät varastoimiaan viittauksia jo muistiin luettuihin tietueisiin. Tällä tavoin peliohjelmisto toimii hieman nopeammin kasvaneen muistin käytön kustannuksella.

Parseriluokasta luodaan ohjelmiston ajon aikana vain yksi instanssi. Luokalla ei ole jäsenmuuttujia, eikä useammasta instanssista näin ollen olisi mitään hyötyä. Nyt luokkaa varten tarvitsee varata muistia vain yhdelle instanssille, eikä kohdelaitteen rajallista muistia käytetä turhaan. Ohjelmiston moduulit, jotka tarvitsevat parseria, saavat parametrina osoittimen tähän instanssiin.

Koska luokasta luodaan joka tapauksessa vain yksi instanssi, se voitaisiin toteuttaa staattisena luokkana tai singleton-luokkana siten, että useamman instanssin luominen siitä olisi mahdotonta. Tällaisen ratkaisun etuna rakenteen alkuperäinen tarkoitus, eli vain yhden instanssin olemassaolo, olisi selkeä lähdekoodia myöhemmin luettaessa. Luokasta ei kuitenkaan haluttu tehdä staattista, koska sen metodien käyttö on suotavaa vain tarkasti rajatuissa tilanteissa lataustaukojen aikana. Staattisen luokan metodeihin olisi pääsy millä moduulilla tahansa milloin tahansa. Tällöin hitaita tiedostonlukuoperaatioita voitaisiin suorittaa vaikkapa kesken pelin intensiivisen toiminnallisen osuuden, mikä ei olisi suotavaa kyseisten operaatioiden suhteellisen hitauden vuoksi. Sama koettiin haitaksi globaaleita vapaita funktioita käytettäessä. Koska TxtSheetParser-luokalla ei ole jäsenmuuttujia tai muuten tallennettavaa tilaa lainkaan, sen ei välttämättä tarvitsisi olla luokka, vaan sen sisältämät metodit voisivat olla vapaita funktioita olematta luokan jäsenmetodeja.

Toiminnallisuuden toteuttaminen luokkana saattaa hämärtää ohjelmiston tämän osuuden tarkoitusperiä ja hämmentää lähdekoodia mahdollisesti myöhemmin tutkivia kehittäjiä, koska sen ei näennäisesti tarvitse olla luokka. Funktioiden keskittäminen luokkaan tällä tavalla kuitenkin auttaa rajoittamaan pääsyä tähän toiminnallisuuteen vain niihin moduuleihin, jotka sitä todella tarvitsevat.

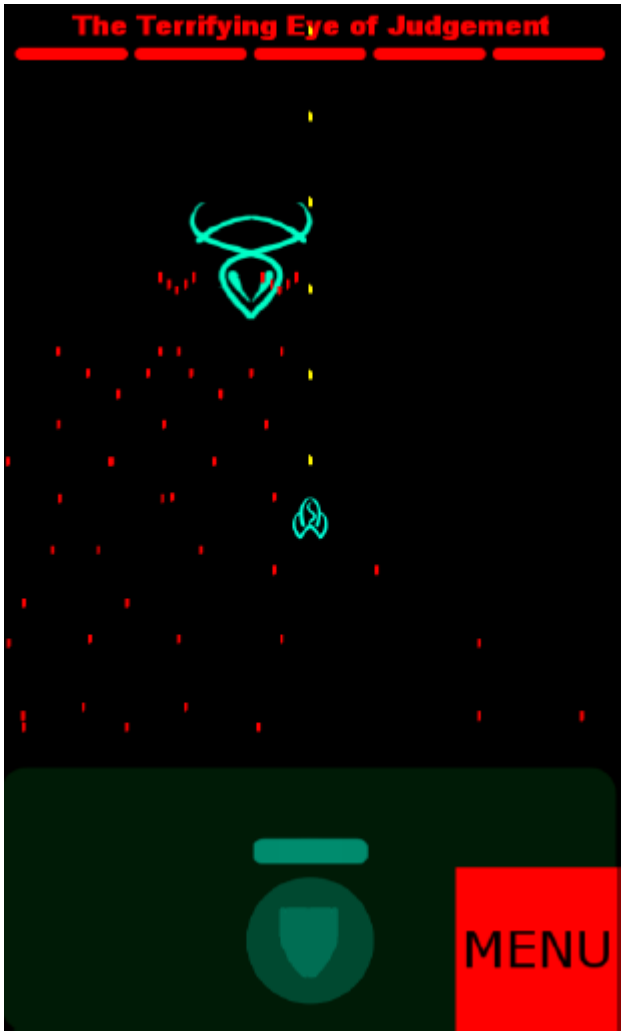
5.5 Projektissa toteutetun pelin kohdealustan vaatimukset

Kohdealustana opinnäytetyöprojektissa kehitetyssä pelissä toimi Android-käyttöjärjestelmä ja sitä käyttävät mobiililaitteet kuten matkapuhelimet ja tablet PC:t. Alustan ominaisuutena kehitettävä ohjelmisto ja kaikki sen tarvitsemat tiedostot oli pakattava binaarimuotoiseen apk-tiedostoon, ja tätä apk-tiedostoa sitten käytettiin asentamaan ohjelmisto kohdelaitteelle.

Seurauksena tästä ominaisuudesta lähdekoodin ulkopuolisten datatiedostojen muokkaus ja muutosten käyttöönotto ohjelmistossa ei ollut aivan niin helppoa kuin alun perin oli toivottu. Kun dataan tehty muutos haluttiin ottaa käyttöön, koko apk-tiedosto jouduttiin luomaan uudelleen PC:llä, poistamaan vanha vastaava tiedosto kohdelaitteesta ja luotu uusi tiedosto asentamaan laitteeseen. Tämä hidasti ohjelmiston kehitysprosessia hieman. Ihannetilanteessa ulkoiseen dataan tehdyt muutokset pystytään toteamaan toimivassa ohjelmistossa hyvin nopeasti ilman pitkällisiä käännös-, pakkaus- ja muunnosoperaatioita, parhaimmillaan jopa ohjelman ollessa käynnissä ilman, että ohjelmistoa tarvitsee sammuttaa ja käynnistää uudelleen.

Hyvänä tämän ominaisuuden vaikutuksena koettiin, etteivät myöskään pelin laitteilleen asentavat pelaajat pääsisi helposti käsiksi pelin sisältämään dataan apk-tiedoston sisälle muokkaamaan pelin ominaisuuksia. Tämän koettiin helpottavan esimerkiksi pelaajia vertailevan pistetaulukon kehittämistä, koska pelaajat eivät pääsisi muokkaamaan peliä omalla laitteellaan helpommaksi kuin muilla ja siten luomaan itselleen muihin pelaajiin nähden helpompaa asemaa päästä pistetaulukon kärkeen. Peliä suunnitellessa tavoitteena ei missään vaiheessa ollut mahdollistaa pelin muokkausta pelaajien toimesta, joten apk-tiedostoksi pakkaamisen jälkeen vaikesti muokattavia datatiedostoja ei koettu negatiivisena seikkana.

6 POHDINTA



Kuvio 4. Kuvankaappaus toteutetun peliohjelmiston prototyypistä.

Opinnäytetyön empiirisen osuuden lopputuotteena valmistui pelattava demo (Kuvio 4.) lopputulokseksi alun perin suunnitellusta pelistä. Demo oli riittävä opinnäytetyön tarpeisiin. Opinnäytetyö priorisoitiin peliä tehdessä korkealle siten, että opinnäytetyön kirjoittamista varten saataisiin tarpeeksi materiaalia vaikka peli ei valmistuisikaan. Valmistunut demo sisälsi tarpeeksi toiminnallisuutta että opinnäytetyön vaatimat ominaisuudet saivat tarvitsemansa kontekstin, jossa niitä voitiin testata ja käyttää.

Tämän opinnäytetyön kannalta oli ongelmallista, että opinnäytetyön aihealue oli vain pieni osa pelin toteutuksesta. Paljon aikaa ja työpanosta jouduttiin käyttämään pelin sellaisiin osa-alueisiin, jotka eivät suoraan hyödyttäneet opinnnäyetyötä. Niiden toteuttaminen kuitenkin

oli välttämätöntä pelin saamiseksi sellaiselle asteelle, että opinnäytetyön tarvitsemat ominaisuudet pystyivät toimimaan merkityksellisessä kontekstissa. Käytännössä opinnäytetyön osuuteen käytettävissä olevaa aikaa oli siis vähemmän kuin koko projektille määritellyt kaksi kuukautta. Ongelman lieventämiseksi projektiin olisi ehkä kannattanut ottaa mukaan lisää henkilöstöä. Toisen ohjelmoijan mukanaolo projektissa olisi mahdollistanut työnjaon sillä tavalla, että opinnäytetyön kirjoittaja olisi pystynyt paremmin keskittymään vain opinnäytetyön käsittelemiin ohjelmiston osiin.

LÄHTEET

Adams, J. 2002. Programming Role Playing Games with DirectX. Course Technology. Saatavissa: http://www.kajak.fi/suomeksi/Kirjasto/E-aineistot. (Luettu Ebrary-tietokannassa 4.10.2012).

Bilas, S. 2002. A Data-Driven Game Object System. Saatavissa: http://scottbilas.com/files/2002/gdc_san_jose/game_objects_slides.pdf. (Luettu 25.5.2011).

Blizzard North. 2000. Diablo 2. Blizzard Entertainment.

Blizzhackers. 2010. [Tutorial] Diablo II Modding. Saatavissa: http://www.blizzhackers.cc/viewtopic.php?t=459161. (Luettu 14.4.2013).

Brownlow, M. 2004. Game Programming Golden Rules. Charles River Media. Saatavissa: http://www.kajak.fi/suomeksi/Kirjasto/E-aineistot. (Luettu Ebrary-tietokannassa 25.5.2011).

DeLoura, M. (toim.) 2000. Game Programming Gems Volume 1. Charles River Media.

Harbour, J. 2004. Game Programming All in One. Premier Press, Incorporated.

Irish, D. 2005. Game Producer's Handbook. Course Technology.

Iron Lore Entertainment. 2006. Titan Quest Modder's Guide. THQ.

McShaffry, M. 2009. Game Coding Complete. Charles River Media.

Patana, T. 2012. Raepuhallusrobotin simulointiympäristö. Kajaanin ammattikorkeakoulu. Opinnäytetyö. Saatavissa: http://urn.fi/URN:NBN:fi:amk-2012060712105 (Luettu 10.4.2013).

Shumaker, S. 2002. Techniques and Strategies for Data-driven design in Game Development. University of Michigan. Saatavissa: http://ai.eecs.umich.edu/soar/Classes/494/talks/Schumaker.pdf. (Luettu 23.2.2013).

Varanese, M. 2002. Game Scripting Mastery. Course Technology.

Wikipedia. 2013. MPQ. Saatavissa: http://en.wikipedia.org/wiki/MPQ. (Luettu 14.4.2013).

Wilson, K. 2002. Data Driven Design. Saatavissa: http://www.gamearchitect.net/Articles/DataDrivenDesign.html. (Luettu 25.5.2011).

Zerbst, S. & Duevel, O. 2004. 3D Game Engine Programming. Charles River Media/Cengage Learning. Saatavissa: http://www.kajak.fi/suomeksi/Kirjasto/E-aineistot. (Luettu Ebrary-tietokannassa 10.4.2013).